EXHIBIT O

Published in Tenable TechBlog

N
Nicholas Miles Follow
Feb 21, 2021 · 20 min read · Listen

Save

Inside SimpliSafe Alarm System

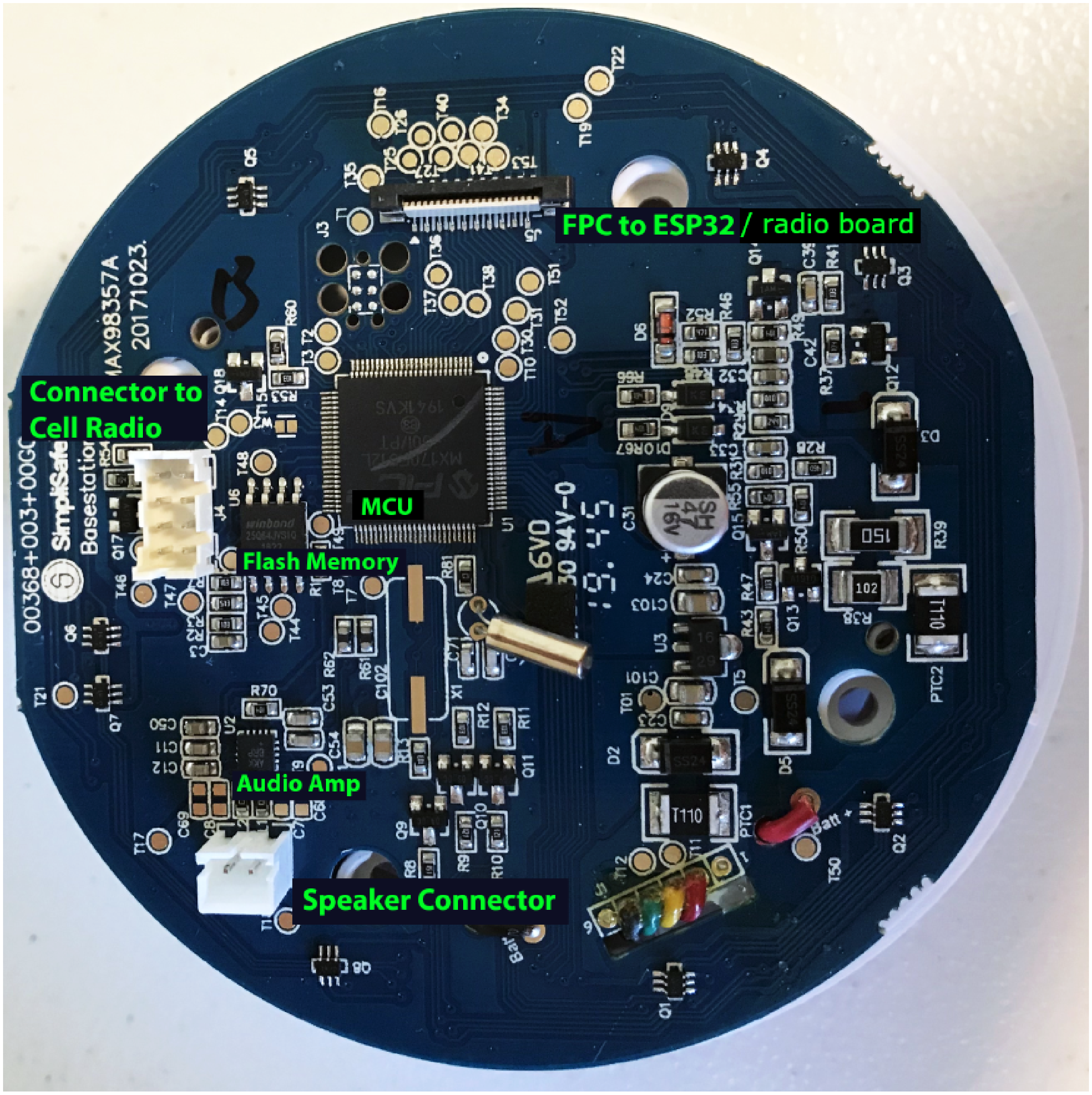
Author: Nick Miles, Co-Author: Chris Lyne

In my last blog (Inside Amazon's Ring Alarm System), I took a deep dive into Amazon's Ring Alarm system. In this blog, I will do the same with a popular competing product — SimpliSafe. In 2016, the infosec firm IOActive released details[1] on how the PIN code sent from the keypad to the base station can be sniffed over the air with a software-defined radio (SDR) and used by an attacker to disarm the system. SimpliSafe's next-generation alarm system, SS3, implements proprietary encryption technology to prevent this attack. In the blog, we will take a look at the new hardware and how they've implemented the encryption.

Dissecting the Hardware

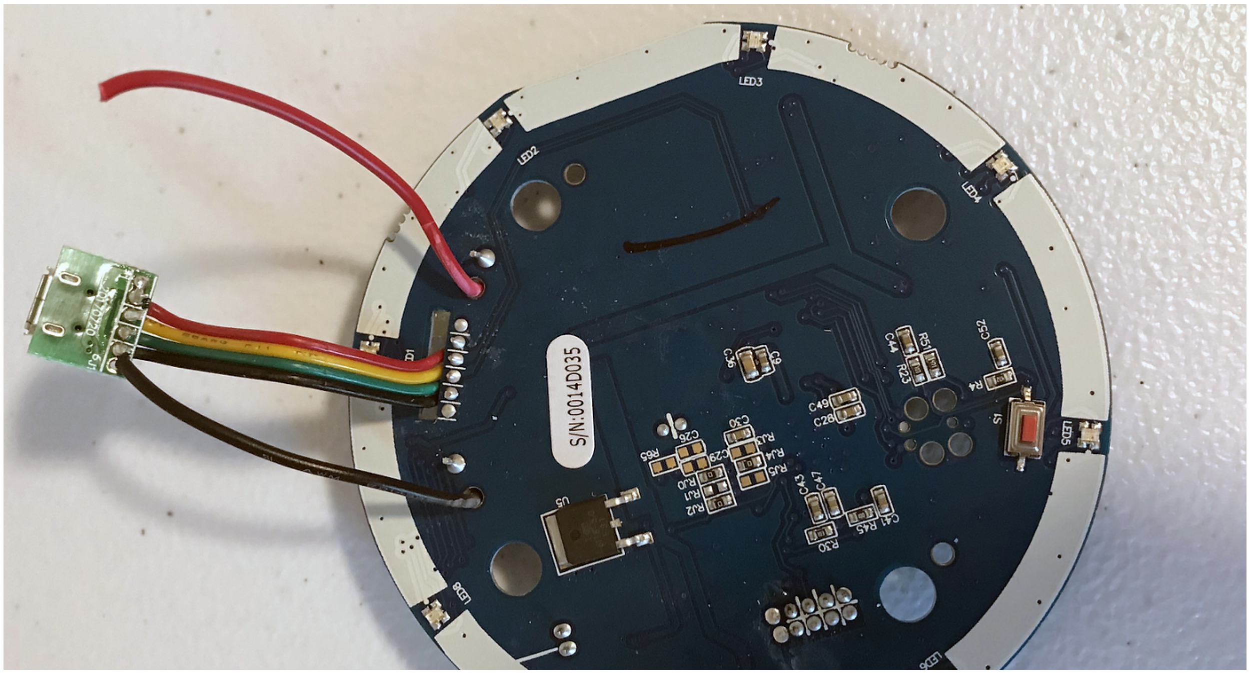
Base Station

Below is the base station mainboard with a PIC Microcontroller (PIC32MX170F512L[11]), flash memory (Winbond 25q64jvsiq), and an audio Amp IC (for the siren).



SimpliSafe SS3 Base Station Circuit Board

Here is the other side of the board with the USB power connector. The red and black wires connect to a 6v rechargeable battery pack (4 AA cells) that provides backup power.



Here is the cellular radio module from Telit:

Cell radio communication backup board (in case the internet goes out!)

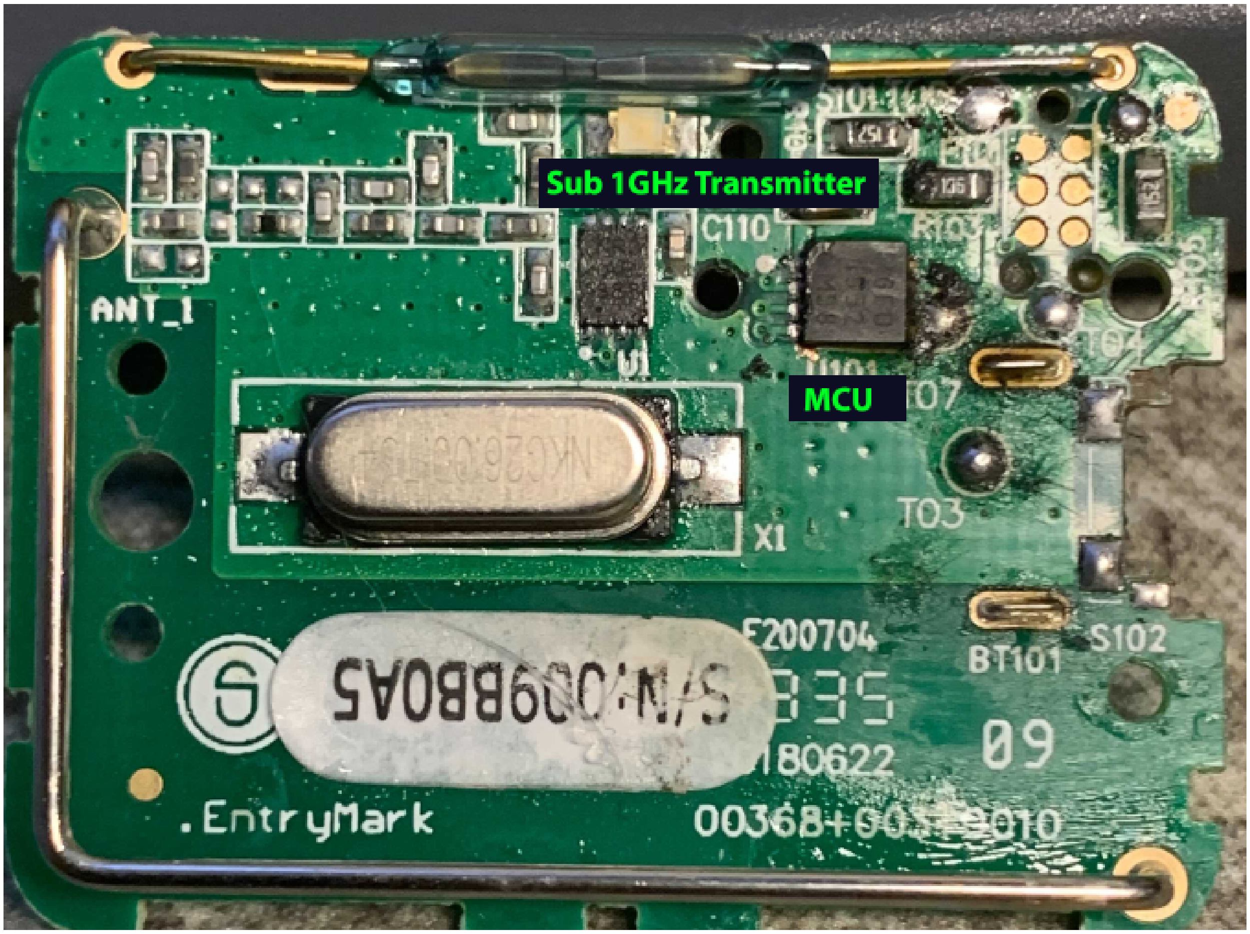
And finally, the radio board, which contains an ESP32 that handles Wi-Fi and Bluetooth Low Energy (BLE), and a general-purpose Sub 1GHz radio IC (Texas Instruments CC1121) for communicating with sensors:

Radio board on the side with RF chip for Sub 1GHz sensor comms

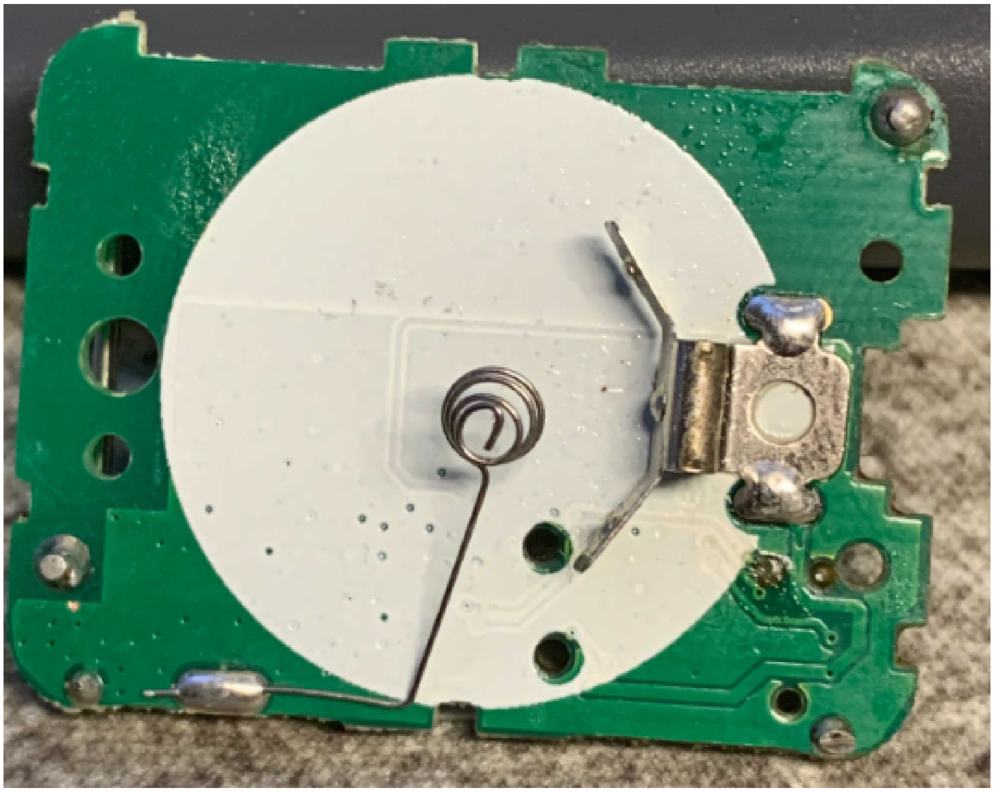
ESP32 for Wifi/BLE on the other side of the radio board

Sensors

Both the entry and motion detectors use a low-power PIC MCU (PIC12LF1572) in tandem with a sub 1GHz transmitter-only IC (SX1243). They are powered with a 3-volt CR2032 battery cell, which makes them very inexpensive to produce. Here are photos of an entry sensor:



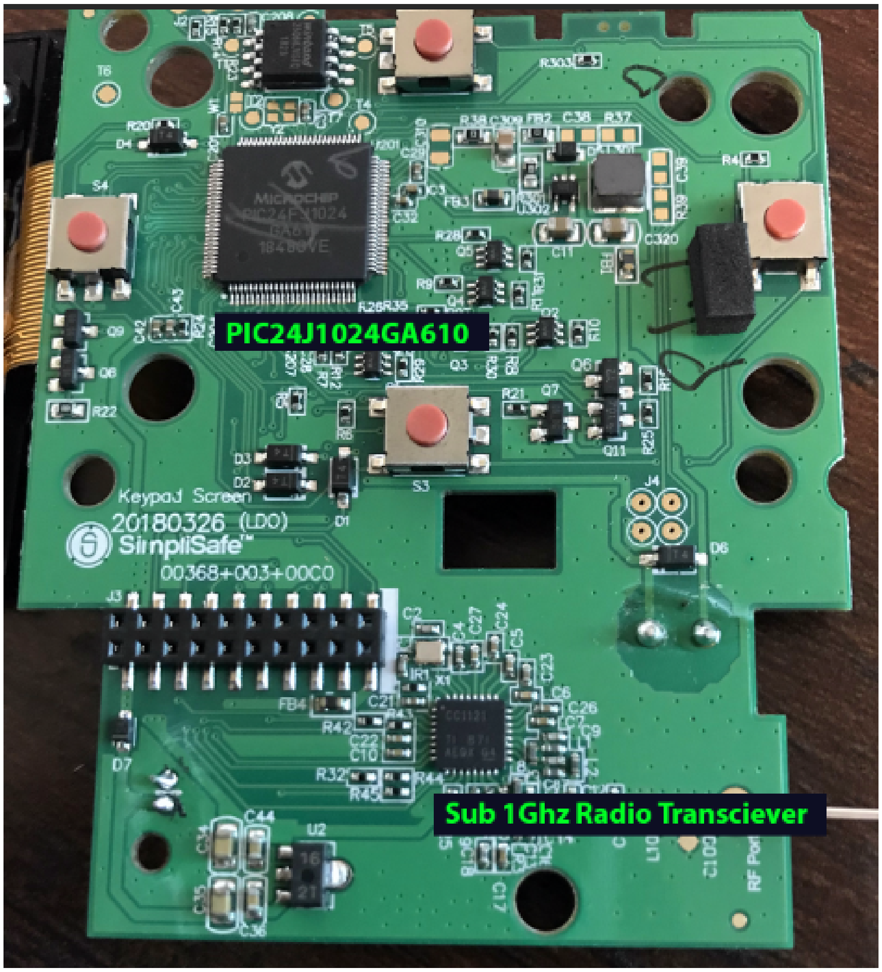
Entry sensor (notice reed switch at the top that is switch via a magnet)



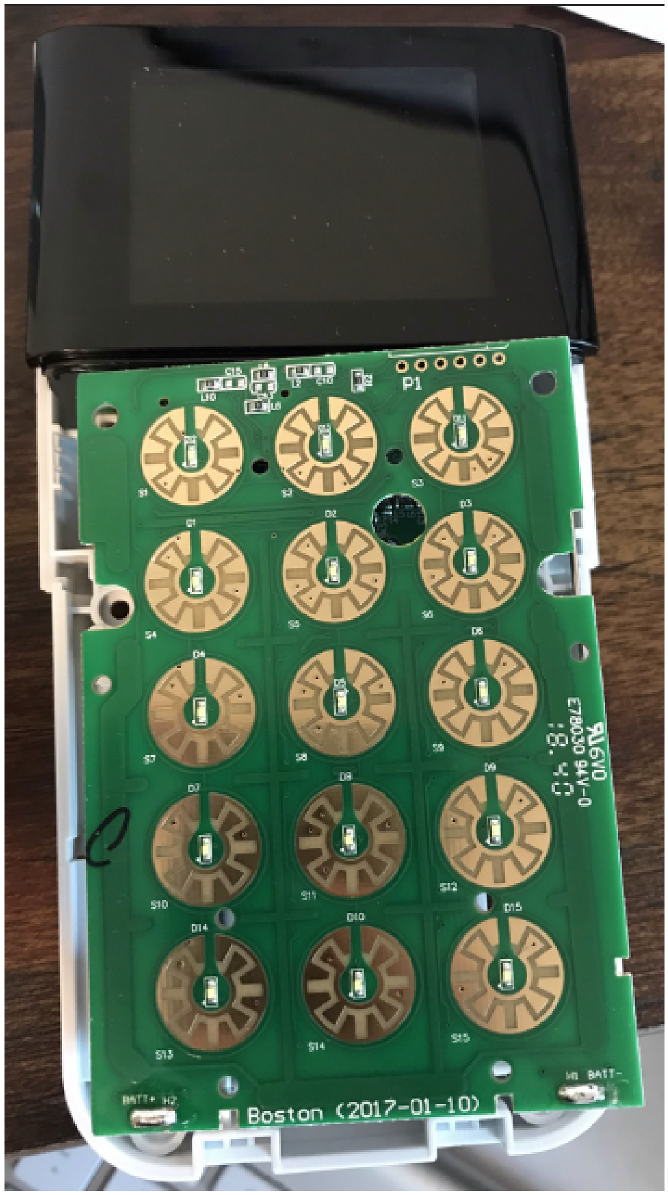
The opposite side of the entry sensor where the CRC2032 battery goes.

Keypad

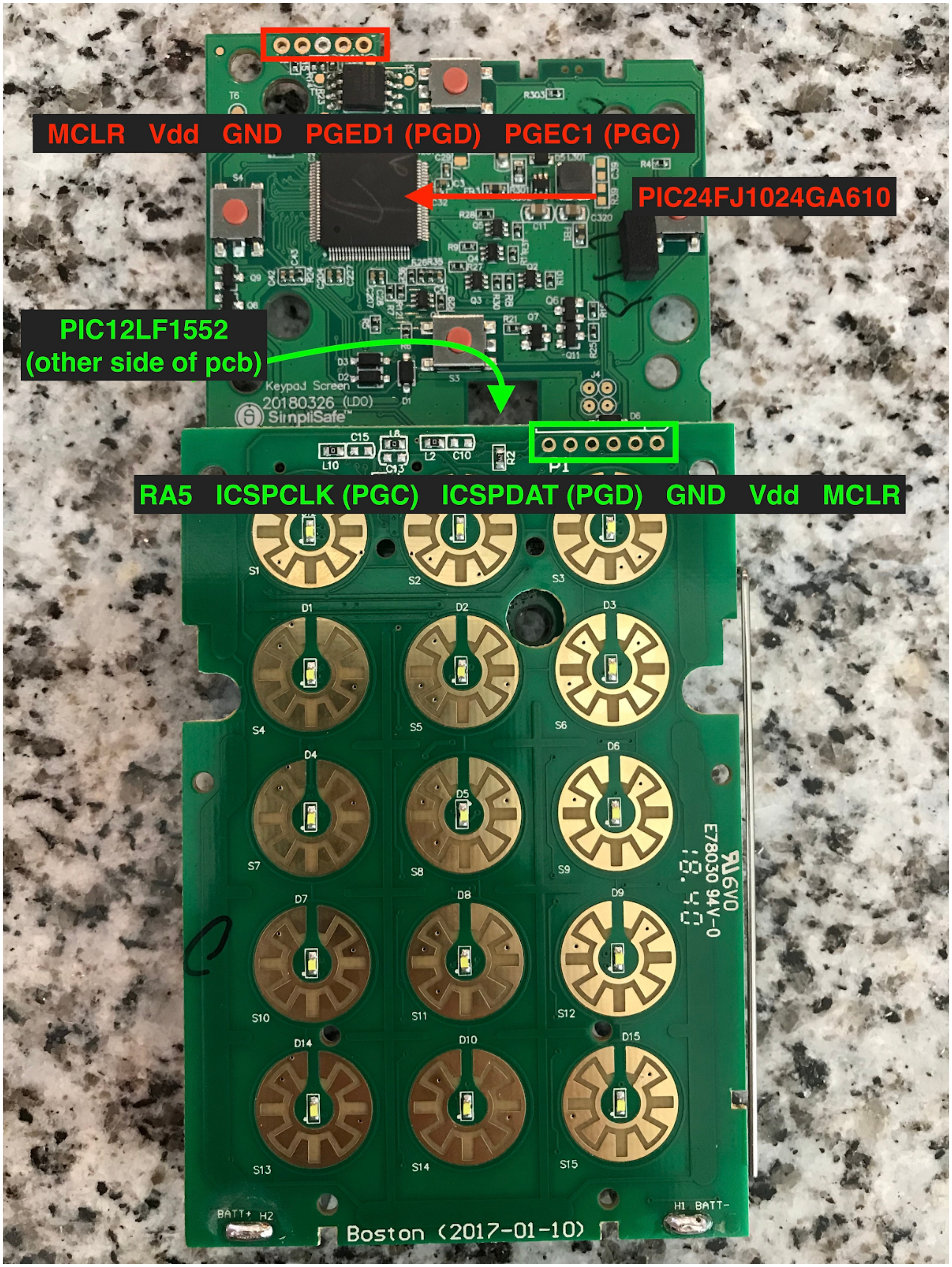
The keypad is very similar in nature to the base station. It uses a PIC microcontroller and the same radio transceiver IC as the base station (CC1121). Below are internal pictures of the keypad and a labeled picture of the programming interface.



Main keypad logic board with PIC MCU and radio chip



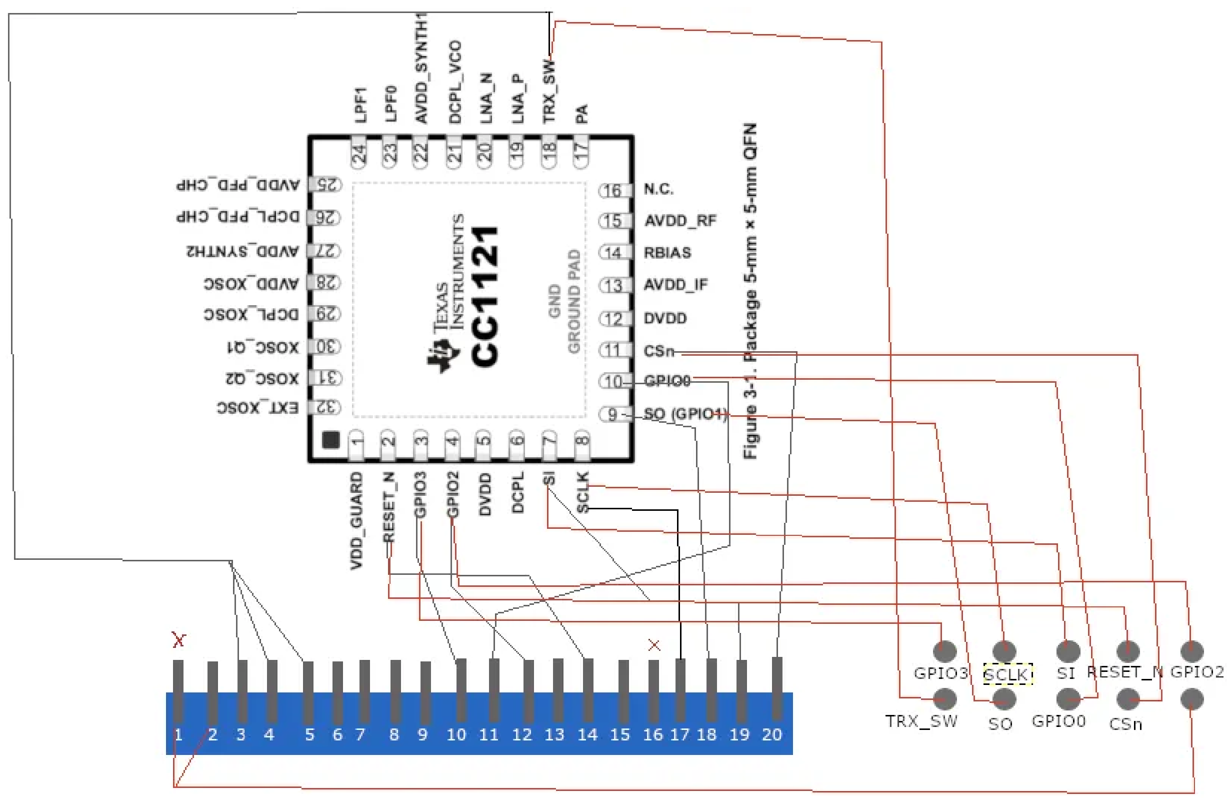
Dome switch PCB for keypad



Programming interfaces for keypad

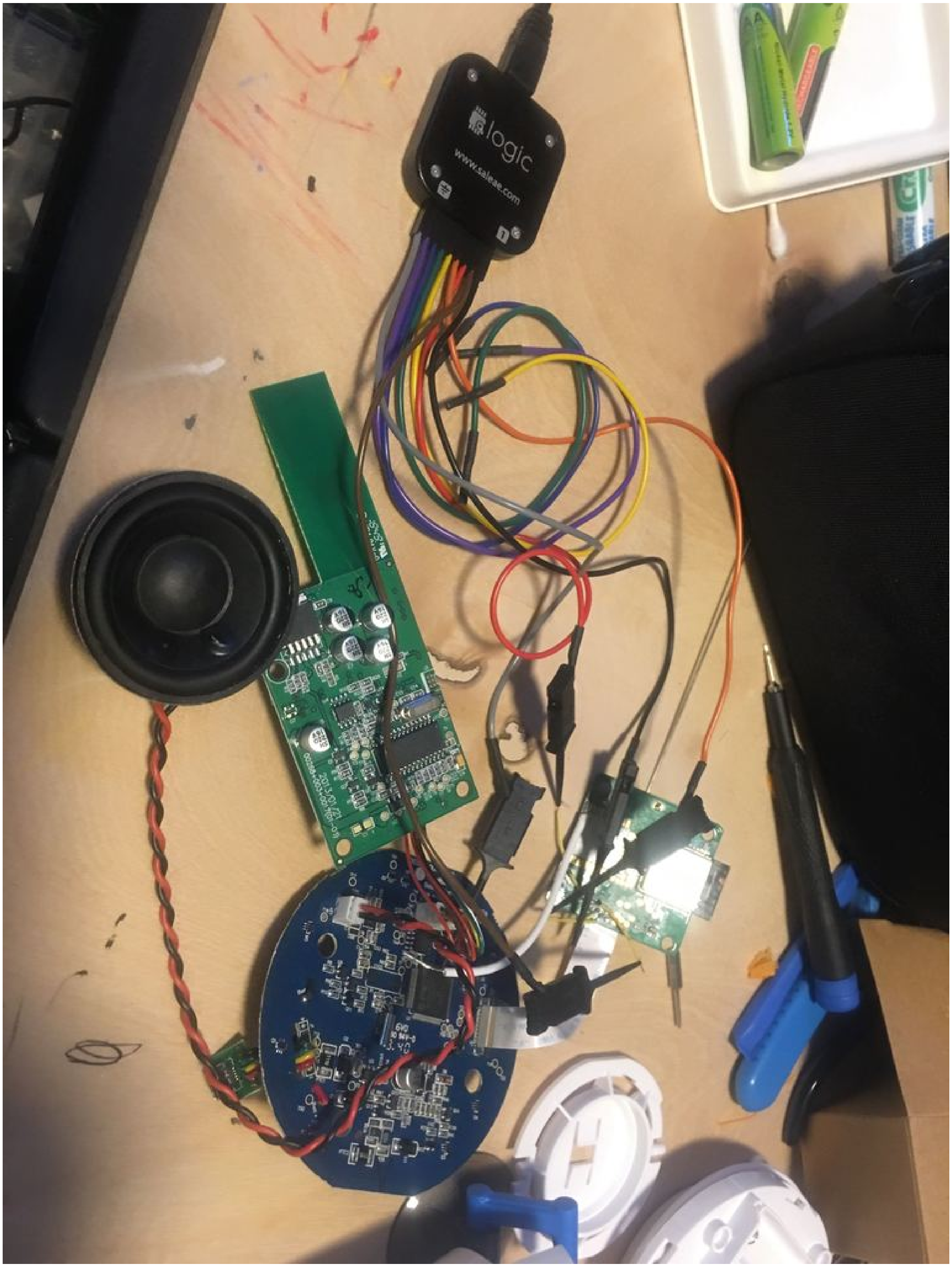
Sub 1GHz Radio Communications

The keypad and base station have Sub 1GHz radio transceiver chips (CC1121). The sensors have transmitter chips. The first thing I did was find a way to tap into the communication between the PIC microcontroller and CC1121 radio IC on the base station so I could monitor it with a logic analyzer. The CC1121 provides SPI (serial peripheral interface[12]) pins for control. We traced out the 4 pins (SCLK [serial clock], SO [serial out], SI [serial in], and CS [chip select]) on the CC1121 back to test points on the board we could easily solder leads to in order to hook up a logic analyzer.



This diagram roughly depicts the PCB for the Simplisafe RF module. The blue connector is a 20-pin 0.5mm pitch FFC connector. To the right, the connections are shown for the 10 test pads. Connections were determined using a continuity test.

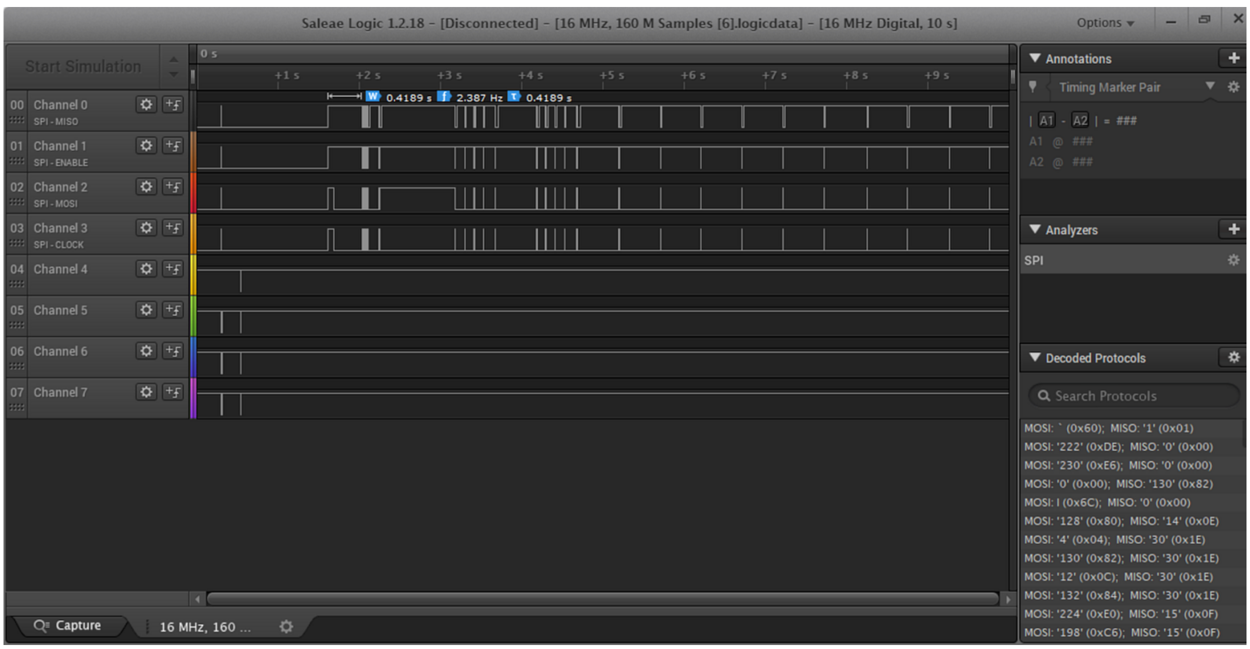
RF Module connection mapping



Logic analyzer hooked up to radio board

Using a Saleae logic analyzer, I was able to decode the raw serial input and output bytes sent between the chips:

SPI Data to Sub 1Ghz Radio Chip



SPI Decoding

In order to understand what the bytes do, you have to dig into various hard-to-find datasheets. Here is the best one I found for our radio chip: https://www.ti.com/lit/ug/swru295e/swru295e.pdf[17]. I eventually ended up writing a decoder that processes the CSV files I can dump from the logic analyzer (https://github.com/tenable/poc/blob/master/SimpliSafe/spi_decoder.py).

Here is an excerpt of the output:

```
Header for packet ID: 0
Header Byte: 0xb0
Access type: Read
Burst Access: False
Address: 0x30
  Command Strobe - SRES - Reset chip

Header for packet ID: 1
Header Byte: 0xef
Access type: Read
Burst Access: True
Address: 0x2f
  Extended Register
    byte read    (0x33[XOSC4]) - 0x41

Header for packet ID: 2
Header Byte: 0xb6
Access type: Read
Burst Access: False
Address: 0x36
  Command Strobe - SIDLE - Exit RX/TX, turn off frequency
synthesizer and exit eWOR mode if applicable

Header for packet ID: 3
Header Byte: 0x40
Access type: Write
Burst Access: True
Address: 0x00
  Regular Register IOCFG3
    byte written (0x00) - 0x02
```

This dump is recording the initialization of the chip after power on. You can see a reset being issued, and various configuration issues being initialized. Here is a snapshot of the registers after configuration:

Regular registers

```
+---------+-----------------+-----------+---------+----------+
| Address |      Name       | Hex Value | Decimal |  Binary  |
+=========+=================+===========+=========+==========+
| 0x00    | IOCFG3          | 0x02      | 2       | 10       |
+---------+-----------------+-----------+---------+----------+
| 0x01    | IOCFG2          | 0x06      | 6       | 110      |
+---------+-----------------+-----------+---------+----------+
| 0x02    | IOCFG1          | 0xb0      | 176     | 10110000 |
```

0x03	IOCFG0	0x40	64	1000000
0x04	SYNC3	0x93	147	10010011
0x05	SYNC2	0x0b	11	1011
0x06	SYNC1	0x51	81	1010001
0x07	SYNC0	0xde	222	11011110
0x08	SYNC_CFG1	0x09	9	1001
0x09	SYNC_CFG0	0x17	23	10111
0x0a	DEVIATION_M	0xaa	170	10101010
0x0b	MODCFG_DEV_E	0x04	4	100
0x0c	DCFILT_CFG	0x15	21	10101
0x0d	PREAMBLE_CFG1	0x18	24	11000
0x0e	PREAMBLE_CFG0	0x2a	42	101010
0x0f	FREQ_IF_CFG	0x3a	58	111010
0x10	IQIC	0x00	0	0
0x11	CHAN_BW	0x03	3	11
0x12	MDMCFG1	0x46	70	1000110
0x13	MDMCFG0	0x05	5	101
0x14	SYMBOL_RATE2	0x63	99	1100011
0x15	SYMBOL_RATE1	0xa9	169	10101001
0x16	SYMBOL_RATE0	0x2a	42	101010
0x17	AGC_REF	0x3c	60	111100
0x18	AGC_CS_THR	0xf8	248	11111000
0x19	AGC_GAIN_ADJUST	0x00	0	0
0x1a	AGC_CFG3	0x91	145	10010001
0x1b	AGC_CFG2	0x20	32	100000

```
+---------+------------------+------------+----------+----------+
| 0x1c    | AGC_CFG1         | 0xa9       | 169      | 10101001 |
+---------+------------------+------------+----------+----------+
| 0x1d    | AGC_CFG0         | 0xc0       | 192      | 11000000 |
+---------+------------------+------------+----------+----------+
| 0x1e    | FIFO_CFG         | 0x46       | 70       | 1000110  |
+---------+------------------+------------+----------+----------+
| 0x1f    | DEV_ADDR         | 0x00       | 0        | 0        |
+---------+------------------+------------+----------+----------+
| 0x20    | SETTLING_CFG     | 0x0b       | 11       | 1011     |
+---------+------------------+------------+----------+----------+
| 0x21    | FS_CFG           | 0x14       | 20       | 10100    |
+---------+------------------+------------+----------+----------+
| 0x22    | WOR_CFG1         | 0x08       | 8        | 1000     |
+---------+------------------+------------+----------+----------+
| 0x23    | WOR_CFG0         | 0x21       | 33       | 100001   |
+---------+------------------+------------+----------+----------+
| 0x24    | WOR_EVENT0_MSB   | 0x00       | 0        | 0        |
+---------+------------------+------------+----------+----------+
| 0x25    | WOR_EVENT0_LSB   | 0x00       | 0        | 0        |
+---------+------------------+------------+----------+----------+
| 0x26    | PKT_CFG2         | 0x04       | 4        | 100      |
+---------+------------------+------------+----------+----------+
| 0x27    | PKT_CFG1         | 0x05       | 5        | 101      |
+---------+------------------+------------+----------+----------+
| 0x28    | PKT_CFG0         | 0x20       | 32       | 100000   |
+---------+------------------+------------+----------+----------+
| 0x29    | RFEND_CFG1       | 0x0f       | 15       | 1111     |
+---------+------------------+------------+----------+----------+
| 0x2a    | RFEND_CFG0       | 0x00       | 0        | 0        |
+---------+------------------+------------+----------+----------+
| 0x2b    | PA_CFG2          | 0x34       | 52       | 110100   |
+---------+------------------+------------+----------+----------+
| 0x2c    | PA_CFG1          | 0x56       | 86       | 1010110  |
+---------+------------------+------------+----------+----------+
| 0x2d    | PA_CFG0          | 0x7e       | 126      | 1111110  |
+---------+------------------+------------+----------+----------+
| 0x2e    | PKT_LEN          | 0xff       | 255      | 11111111 |
+---------+------------------+------------+----------+----------+
```

Extended Registers

```
+---------+-----------------+-----------+---------+----------+
| Address |      Name       | Hex Value | Decimal |  Binary  |
+=========+=================+===========+=========+==========+
| 0x00    | IF_MIX_CFG      | 0x0       | 0       | 0        |
+---------+-----------------+-----------+---------+----------+
```

```
| 0x01     | FREQOFF_CFG    | 0x30      | 48      | 110000   |
+----------+----------------+-----------+---------+----------+
| 0x02     | TOC_CFG        | 0x4b      | 75      | 1001011  |
+----------+----------------+-----------+---------+----------+
| 0x0a     | FREQOFF1       | 0x0       | 0       | 0        |
+----------+----------------+-----------+---------+----------+
| 0x0b     | FREQOFF0       | 0xde      | 222     | 11011110 |
+----------+----------------+-----------+---------+----------+
| 0x0c     | FREQ2          | 0x6c      | 108     | 1101100  |
+----------+----------------+-----------+---------+----------+
| 0x0d     | FREQ1          | 0x7a      | 122     | 1111010  |
+----------+----------------+-----------+---------+----------+
| 0x0e     | FREQ0          | 0xe1      | 225     | 11100001 |
+----------+----------------+-----------+---------+----------+
| 0x91     | SERIAL_STATUS  |           |         |          |
+----------+----------------+-----------+---------+----------+
| 0x12     | FS_DIG1        | 0x0       | 0       | 0        |
+----------+----------------+-----------+---------+----------+
| 0x13     | FS_DIG0        | 0x5f      | 95      | 1011111  |
+----------+----------------+-----------+---------+----------+
| 0x14     | FS_CAL3        |           |         |          |
+----------+----------------+-----------+---------+----------+
| 0x15     | FS_CAL2        | 0x20      | 32      | 100000   |
+----------+----------------+-----------+---------+----------+
| 0x16     | FS_CAL1        | 0x40      | 64      | 1000000  |
+----------+----------------+-----------+---------+----------+
| 0x17     | FS_CAL0        | 0xe       | 14      | 1110     |
+----------+----------------+-----------+---------+----------+
| 0x18     | FS_CHP         | 0x28      | 40      | 101000   |
+----------+----------------+-----------+---------+----------+
| 0x19     | FS_DIVTWO      | 0x3       | 3       | 11       |
+----------+----------------+-----------+---------+----------+
| 0x1b     | FS_DSM0        | 0x33      | 51      | 110011   |
+----------+----------------+-----------+---------+----------+
| 0x1d     | FS_DVC0        | 0x17      | 23      | 10111    |
+----------+----------------+-----------+---------+----------+
| 0x1f     | FS_PFD         | 0x50      | 80      | 1010000  |
+----------+----------------+-----------+---------+----------+
| 0x20     | FS_PRE         | 0x6e      | 110     | 1101110  |
+----------+----------------+-----------+---------+----------+
| 0x21     | FS_REG_DIV_CML | 0x14      | 20      | 10100    |
+----------+----------------+-----------+---------+----------+
| 0x22     | FS_SPARE       | 0xac      | 172     | 10101100 |
+----------+----------------+-----------+---------+----------+
| 0x23     | FS_VCO4        | 0x11      | 17      | 10001    |
+----------+----------------+-----------+---------+----------+
| 0x24     | FS_VCO3        |           |         |          |
+----------+----------------+-----------+---------+----------+
| 0x25     | FS_VCO2        | 0x48      | 72      | 1001000  |
+----------+----------------+-----------+---------+----------+
```

```
| 0x26     | FS_VCO1         |           |          |          |
+----------+-----------------+-----------+----------+----------+
| 0x27     | FS_VCO0         | 0xb4      | 180      | 10110100 |
+----------+-----------------+-----------+----------+----------+
| 0x28     | GBIAS6          |           |          |          |
+----------+-----------------+-----------+----------+----------+
| 0x29     | GBIAS5          |           |          |          |
+----------+-----------------+-----------+----------+----------+
| 0x2a     | GBIAS4          |           |          |          |
+----------+-----------------+-----------+----------+----------+
| 0x2b     | GBIAS3          |           |          |          |
+----------+-----------------+-----------+----------+----------+
| 0x2c     | GBIAS2          |           |          |          |
+----------+-----------------+-----------+----------+----------+
| 0x2d     | GBIAS1          |           |          |          |
+----------+-----------------+-----------+----------+----------+
| 0x2e     | GBIAS0          |           |          |          |
+----------+-----------------+-----------+----------+----------+
| 0x2f     | IFAMP           |           |          |          |
+----------+-----------------+-----------+----------+----------+
| 0x30     | LNA             |           |          |          |
+----------+-----------------+-----------+----------+----------+
| 0x31     | RXMIX           |           |          |          |
+----------+-----------------+-----------+----------+----------+
| 0x32     | XOSC5           | 0xe       | 14       | 1110     |
+----------+-----------------+-----------+----------+----------+
| 0x33     | XOSC4           |           |          |          |
+----------+-----------------+-----------+----------+----------+
| 0x34     | XOSC3           |           |          |          |
+----------+-----------------+-----------+----------+----------+
| 0x35     | XOSC2           |           |          |          |
+----------+-----------------+-----------+----------+----------+
| 0x36     | XOSC1           | 0x3       | 3        | 11       |
+----------+-----------------+-----------+----------+----------+
```

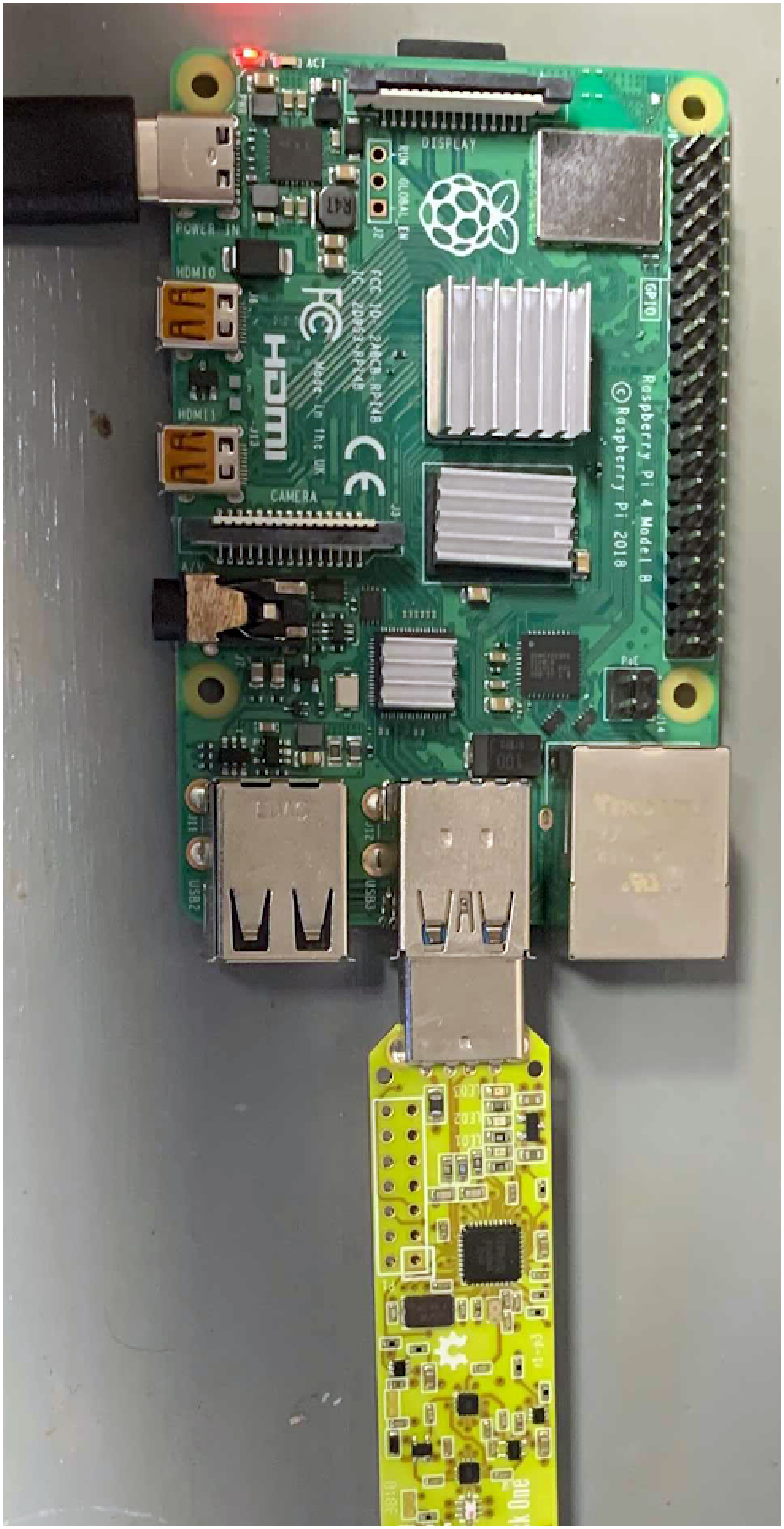
With these configuration parameters understood, it is now possible to configure a software-defined radio like the HackRF[13] or YARD Stick One[14] to receive and decode SimpliSafe packets.

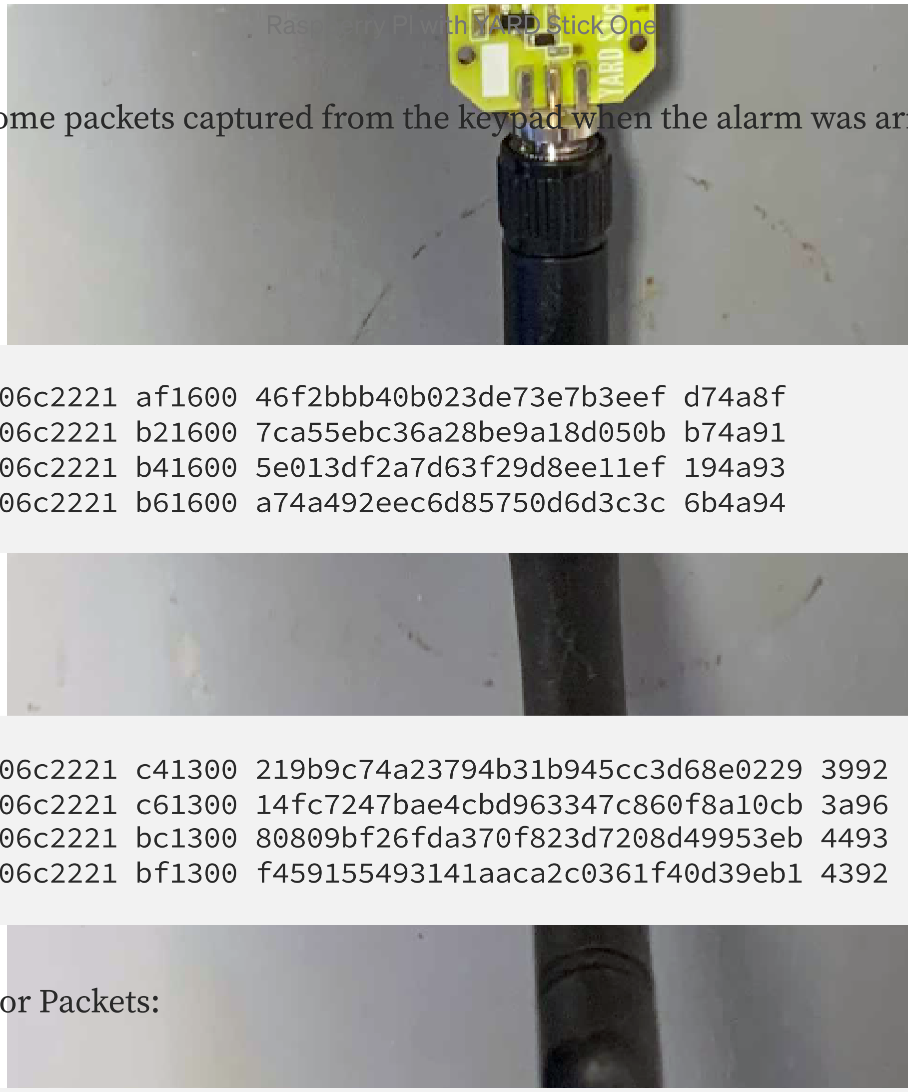
Sniffing Wireless Sensor/Keypad Traffic

Using a YARD Stick One, and rfcat[15] and the parameters discovered in the steps above, a python script can be used to obtain packets:

```
d = RfCat()

d.setFreq(433899963)
d.setMdmModulation(MOD_2FSK)
d.setMdmSyncWord(0x930b)
d.setMdmSyncMode(SYNCM_16_of_16)
d.setMdmSyncMode(SYNCM_CARRIER_16_of_16)
d.setMdmDRate(4800)
d.makePktFLEN(0xff)

print("Press <enter> to stop")

while not keystop():
  try:
    pkt,t = d.RFrecv()
```

Python rfcat code excerpt


ACT
DISPLAY
POWER IN
HDMI0
HDMI1
CAMERA
GPIO
Raspberry Pi 4 Model B
© Raspberry Pi 2018
PoE



Raspberry Pi with YARD Stick One

Here are some packets captured from the keypad when the alarm was armed or disarmed.

Home:

```
15 01 006c2221 af1600 46f2bbb40b023de73e7b3eef d74a8f
15 01 006c2221 b21600 7ca55ebc36a28be9a18d050b b74a91
15 01 006c2221 b41600 5e013df2a7d63f29d8ee11ef 194a93
15 01 006c2221 b61600 a74a492eec6d85750d6d3c3c 6b4a94
```

Disarm:

```
18 01 006c2221 c41300 219b9c74a23794b31b945cc3d68e0229 3992
18 01 006c2221 c61300 14fc7247bae4cbd963347c860f8a10cb 3a96
18 01 006c2221 bc1300 80809bf26fda370f823d7208d49953eb 4493
18 01 006c2221 bf1300 f459155493141aaca2c0361f40d39eb1 4392
```

Entry Sensor Packets:

```
16 02 007289cb f9d202 6b2d82a0f26ded4a3d95ae006723 fa51
16 02 007289cb fad202 7d441c349698ab938ee973c0cd9f 098d
16 02 007289cb fed202 0994ea1ac9155f6076ff75105bb8 2b24
16 02 007289cb 06d302 65332d08bc4cc2df10a823039eef e8cf
```

Panic Button Press:

```
16 02 006cdc17 431f00 ccdbd36b5e9c9afab6f25247d7cd 0cca
```

Keyfob Packets:

```
16 02 0070f8bb 355a00 c991a297447c3b5f9896460c2a0a 378f
16 02 0070f8bb 365a00 619f3a672661563d95ecdec007a5 388f
```

After capturing enough packets, a pattern becomes apparent. Here is the breakdown of packet contents from my initial analysis.

Packet Example:

```
16 02 007289cb f9d202 6b2d82a0f26ded4a3d95ae006723 fa51

Size                 : 16 (22 bytes, not including length byte)
  Packet Flag / Type?  : 02
  Device Serial        : 007289cb
  Encrypted Data?      : 6b2d82a0f26ded4a3d95ae006723
  CheckSum             : fa51
```

I also noticed that pressing the "Test" button on sensors caused them to send a special packet. Note that this button is used to "bind" (or "pair") the sensor to the base station. Here are a few from the entry sensors:

```
16 f2 006f0538 05 fc40f8a66fd761ce2e2edf54ab0a5404 316c
16 f2 005f5dc9 05 2a0220efdd4598e89171ab4170c7bbe3 7615
16 f2 007289cb 05 e6af9ac41e4cde7407efc5a28334e04a cb76
```

What's interesting is that the panic button doesn't have a separate bind button, so it sends its bind packet in addition to its regular data packet every time it's triggered. Since it's only ever pressed in an emergency, there would never be a way to capture the bind packet after the initial bind without triggering the alarm, unless you trigger it and prevent the signal from reaching the base station (e.g. by using a faraday cage / bag).

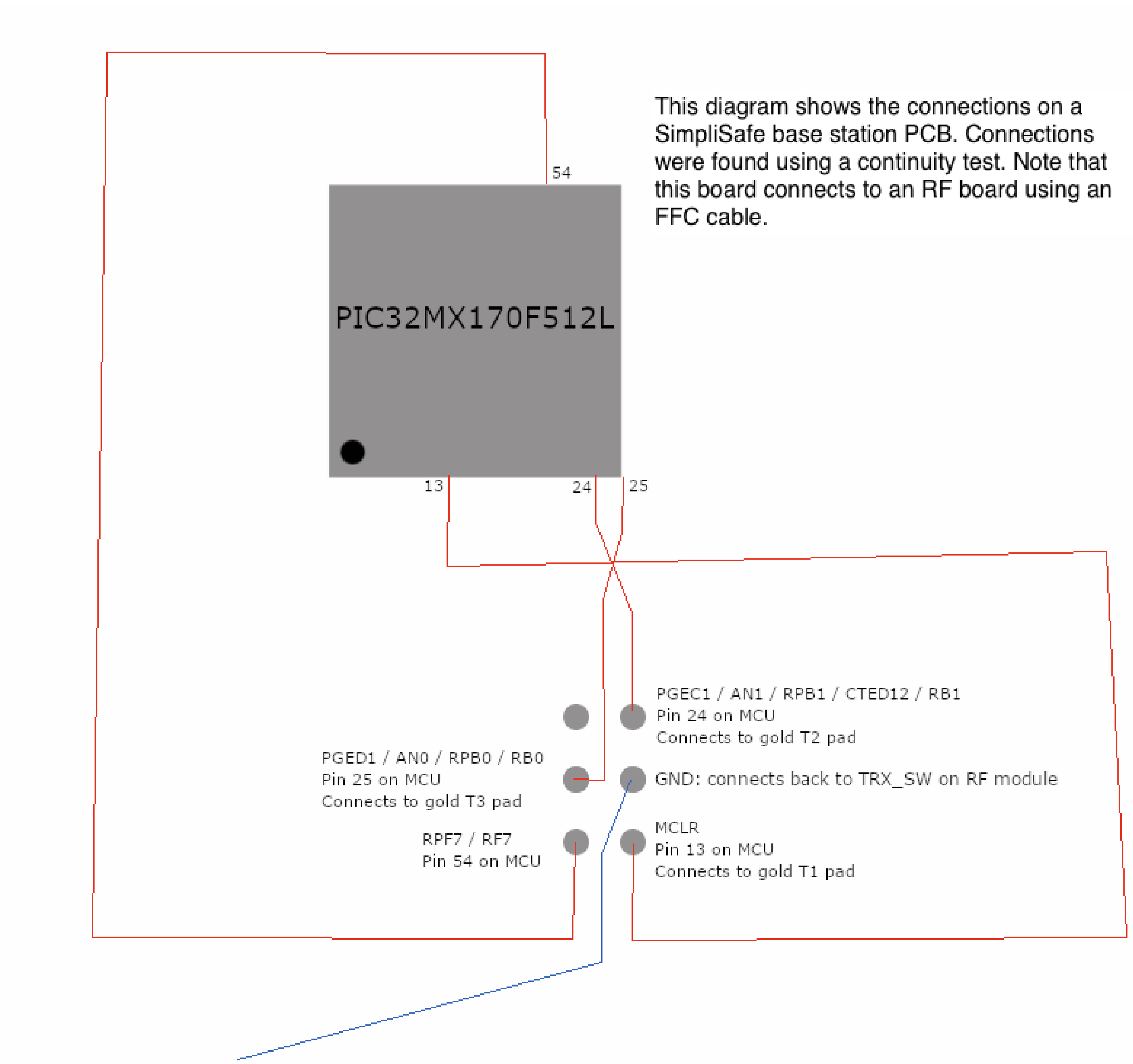
```
16 f2 006cdc17 03 f3dff6f2bdb17c9250462a7de968fc74 f4f6
```

The packets are always the same each time a bind is triggered. They never change. The packets include a length field (0x16), a flag (0xf2) indicating that the packet is for binding, the serial number of the device, what looks to be a constant, some other seemingly random potentially-encrypted data, and a two-byte checksum at the end.

Without digging further into the microcontrollers themselves, it's hard to determine much more about what's going on. It's obvious that the pin data for arming and disarming is obfuscated/encrypted in some way. So I decided to turn my attention towards the PICs to see if I can pull firmware.

Attacking Protected PIC MCUs

There are programming pins on the base station, keypad, and sensors for attaching a programmer. I used a PICkit 3 for attempting to extract firmware from these microcontrollers. Here is the pinout for connecting a programmer to the base station:



PIC Programmer pin mapping (base station)

PIC Programming Pins (Base Station):

Pin Description	MCU Pin	Base Station Test Pad
PGED1	25	T3
PGEC1	24	T2
GND	46	T46
VDD	2	T47
MCLR	13	T1

Using the PICkit I tried to read the device but found that the device has code protection enabled:

```
-
Target voltage detected
Target device PIC32MX170F512L found.
Device ID Revision = A0

Reading...

The following memory area(s) will be read:
program memory: start address = 0x1d000000, end address = 0x1d07ffff
boot config memory
configuration memory
The device is code protected.
Failed to read device
Selected device and target: memory mismatch.
```

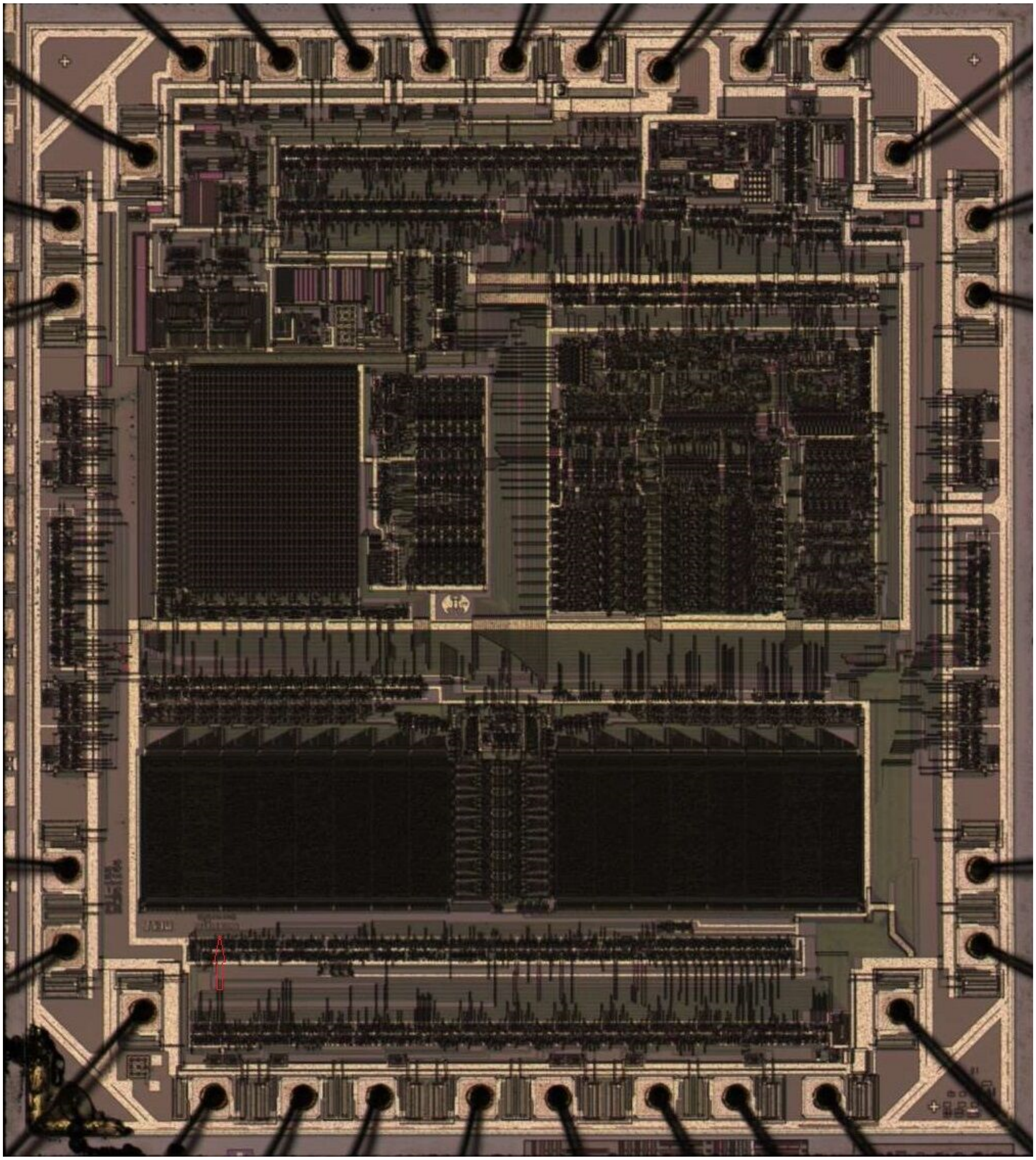
Attempting to read PIC, but code protected :(

I also tried reading MCU code from the entry, keypad, and motion sensors. All were code protected.

Breaking PIC Code Protection

After doing a bit of research, there are three general approaches that may be used to unlock Microcontrollers. All three require an IC which has been decapped. Decapping is a process where the packaging covering the silicon die (either plastic, ceramic, or epoxy) is removed to expose it. There are a variety of methods to accomplish this, including laser etching and chemical etching.

I sent one of the base station MCUs to a lab, and below is an image of the decapped chip they sent back.



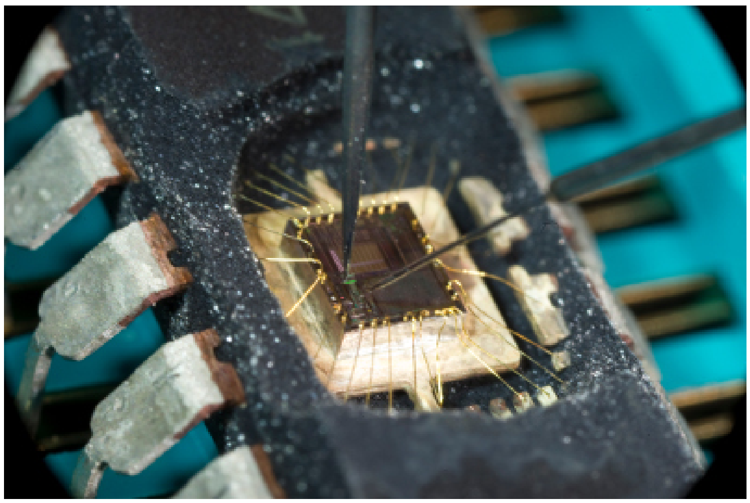
With the chip “functionally” decapped (meaning it still works after having the package etched away), there are various ways I’m aware of in which the copy protection can be attacked.

UV Light Exposure

The is a blog[2] I found that goes into detail about how some PIC devices can be unlocked by exposing the memory of the security bits to UV light which clears them. This works on a variety of PIC microcontrollers and is the method in which I believe a lab unlocked the PIC entry sensor; I'll go into detail further down.

Micro Probing

By using a micro probe station, one can precisely position microscopic probes using a microscope. It is sometimes possible to tap certain traces to read data while it is being retrieved from memory. This is like running WireShark on an IC.



Micro Probe station in action

Focused Ion Beam (FIB) Mill

This technology uses an extremely focused gallium ion beam to either add (deposit) or remove (ablate) material from a silicon die. This can be used to edit circuits on a silicon die and is commonly used during the development/prototyping and testing/debugging of integrated circuits. FIB is used in combination with SEM (scanning electron microscopy) in order to monitor/control the milling process. The consumables and equipment rental costs for this process are fairly expensive, but with chips using smaller and smaller lithography processes, this is the only way to attack the protection on more advanced chips, such as the PIC32 used in the SimpliSafe base station.

Fabrication Labs

I found that the cheapest and easiest way to try the above-mentioned attacks on microcontrollers was by contacting various labs in that specialize in reverse engineering both PCBs and extracting code from MCUs. After contacting various labs and getting quotes, I sent ICs from entry sensors and the base station.

An attempt was made on the base station chip using FIB, but it failed. They tried a second time, said it was successful, and sent me code designed to run on the same chip, but it was obviously from another product. It appeared that my order had been mixed up with someone else's. At this point, I gave up on the effort for the base station chip, as it was starting to get expensive.

For the alarm sensors, however, a different lab was able to extract the code of the chip for under $1k. Therefore, they used one of the methods (other than FIB due to the cost) described above to extract the code, likely UV light exposure to reset the security fuses. They even offered to sell me the technology to DIY :)

hello

my friend, thanks for you!

Congratulations on your success. It is not accidental for us to unlock this chip. It is inevitable because of our 18 years of experience and technology accumulation!

Regarding how you mentioned how to crack, I am sorry to say to you: For every chip model series, our engineers spend a lot of money on cracking research. Some models may not ultimately succeed even if they cost a lot of money.

Therefore, if you want to learn this series of chip decryption solutions, we can provide you with plans and drawings; and the equipment used! So you can master this series of chip cracking. But the cost is very high. In simple terms, it is a Technology transfer process!

I'm so glad you helped me share our company with your colleagues. Give them our website: [redacted]

thanks again! Looking forward to our cooperation next time!

Tech Transfer Offer

I was able to reverse engineer the extracted code and determine that it contains a hard-coded AES key. In addition to this key, it was found that the binding key is hard-coded into the chip. Therefore, it stands to reason that if the base station code is ever compromised it should be possible to decrypt the binding packets and obtain the hardcoded AES keys needed to decode the communication with any SS3 device.

I implemented a decoding tool for the entry sensor I had from reverse-engineering the code. It will only work with sensors you have AES keys for. You can download the code here[16]. I programmed firmware with the known AES key onto a fresh SimpliSafe entry sensor and tested it out.

Entry sensor with pirate firmware :)

```
************************************************************************
Packet Recieved:
1602007289cba7b1019fdc5efa00f8723221f9f407ad83bdf3
=== PACKET DISSECTION ===
Packet          : 1602007289cba7b1019fdc5efa00f8723221f9f407ad83bdf3
Length          : 16 (22)
Serial          :     007289cb
Counter         :             a7b101
CMAC:           :                   9fdc5efa
Encrypted Data  :                           00f8723221f9f407ad83
Chksum          :                                               bdf3

Data Decryption:
  AES Call 1 Res : 059074322196f4079282107362a5c6e1
  Decrypted Data : 05680600006f00003f01

CMAC Verify:
  AES Call 2 Res : 34d21f79bc403b8ccbaa7fb1585be8e2
  LSFR Res1: 4d3c72db5d1422f8c3b06d556c043c86
  CHK Data1 : 00f8723221f9f407ad83
  CHK Data2 : 4dc400e97cedd6ff6e336d556c043c86
  LSFR Res2: e2db57edef279283839be71b395681b6
  LSFR Res2_xor: e2db57edef279203839be71b395681e6
  LSFR Res3: 2b7d136be359a90d298bcd9fc6fe42b4
  AES Call 3 Res : b4a14d91c09e5e98d24f67c98e787f88
  CHK Data3 (CALCULATED CMAC): 9fdc5efa
     CMAC Match!

Checksum Verify:
  Calculated Checksum:bdf3
     Checksum Verified!
************************************************************************
```

Packet decoder in action

As you can see, it works! The way attached devices use hardcoded keys explains some of the attacks we found using a rogue keypad. This can allow an attacker with physical access, without knowledge of the PIN, the ability to pair a rogue keypad and subsequently disarm the alarm system[3] [4]. (https://www.youtube.com/watch?v=YIfc_jQAB9g, https://www.youtube.com/watch?v=So4fzBzxbu8). With sensors that have bi-directional communication, a key exchange could take place to 100% prevent these sorts of attacks without having to rely on complicated logic on the base station to protect against it.

ESP32

As mentioned previously, the SimpliSafe base station has a radio board connected to it via an FPC cable. The radio board provides Wi-Fi and BLE connectivity to the main

base station board via an ESP32 SoC. A cloud connection is required for firmware updates and remote control.

ESP32

With the hopes of finding some juicy code on the ESP32, a large chunk of time was spent reverse engineering the firmware format. The goal was to be able to take a flash dump from an ESP32 and then analyze it in IDA Pro. The problem was that there wasn't any tooling available that could do exactly this. So we created our own. We'll next talk about our RE process from a high level.

For a frame of reference, when we first dumped the flash on an ESP32, the 'file' utility had no idea what it was looking at.

```
sh-3.2$ file esp32dump.bin
esp32dump.bin: data
```

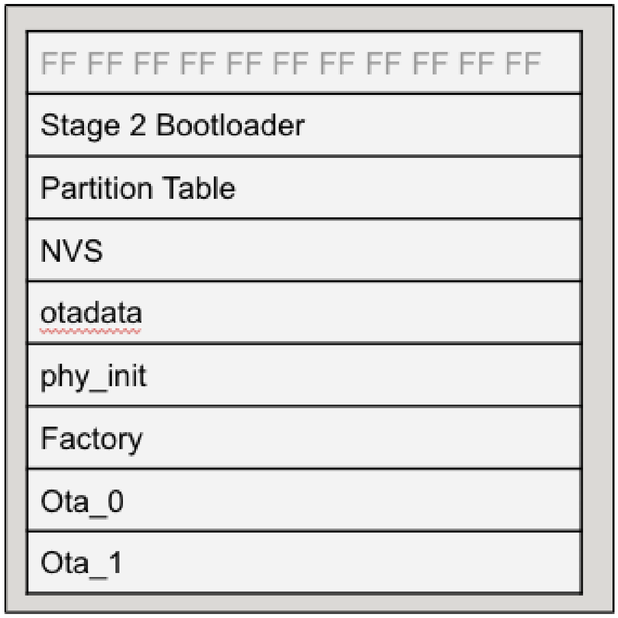
Binwalk wasn't much help either. It identified some Unix paths, certificates, private keys, and maybe some SHA256 constants. But, really, this wasn't much help. We were interested in the **code**.

```
sh-3.2$ binwalk esp32dump.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
8272          0x2050          Unix path: /home/darrell/git/esp-idf/components/soc/esp32/rtc_clk.c
332736        0x513C0         Unix path: /mnt/ss3_fw_source/esp-idf/components/esp32/./crosscore_int.c
332968        0x514A8         Unix path: /mnt/ss3_fw_source/esp-idf/components/esp32/./heap_alloc_caps.c
333784        0x517D8         Unix path: /mnt/ss3_fw_source/esp-idf/components/esp32/./intr_alloc.c
335468        0x51E6C         Unix path: /mnt/ss3_fw_source/esp-idf/components/freertos/./heap_regions.c
336444        0x5223C         Unix path: /mnt/ss3_fw_source/esp-idf/components/freertos/./timers.c
336900        0x52404         Unix path: /mnt/ss3_fw_source/esp-idf/components/newlib/./locks.c
337320        0x525A8         Unix path: /mnt/ss3_fw_source/esp-idf/components/soc/esp32/rtc_clk.c
337808        0x52790         Unix path: /mnt/ss3_fw_source/esp-idf/components/spi_flash/./spi_flash_rom_patch.c
338692        0x52B04         Unix path: /mnt/ss3_fw_source/esp-idf/components/vfs/./vfs.c
351836        0x55E5C         Unix path: /mnt/ss3_fw_source/main/./device_nv.c
353121        0x56361         PEM certificate
354828        0x56A0C         Unix path: /mnt/ss3_fw_source/esp-idf/components/app_update/./esp_ota_ops.c
357656        0x57518         Unix path: /mnt/ss3_fw_source/components/at_core/./at_port.c
360740        0x58124         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/osi/future.c
361188        0x582E4         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/btc/core/btc_ble_storage.c
361480        0x58408         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/btc/core/btc_config.c
393096        0x5FF88         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/stack/btu/btu_task.c
450480        0x6DFB0         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/bta/sys/bta_sys_main.c
451348        0x6E314         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/btcore/bdaddr.c
452656        0x6E830         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/device/controller.c
454708        0x6F034         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/hci/hci_layer.c
455104        0x6F1C0         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/hci/hci_packet_factory.c
456124        0x6F5BC         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/hci/packet_fragmenter.c
457124        0x6F9A4         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/osi/alarm.c
457728        0x6FC00         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/osi/config.c
458580        0x6FF54         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/osi/fixed_queue.c
472420        0x73564         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/stack/btm/btm_ble_bgconn.c
503480        0x7AEB8         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/device/interop.c
504108        0x7B12C         Unix path: /mnt/ss3_fw_source/esp-idf/components/bt/bluedroid/hci/hci_hal_h4.c
506032        0x7B8B0         Unix path: /mnt/ss3_fw_source/esp-idf/components/driver/./rtc_module.c
511336        0x7CD68         Unix path: /mnt/ss3_fw_source/esp-idf/components/driver/./uart.c
518036        0x7E794         Unix path: /mnt/ss3_fw_source/esp-idf/components/driver/./gpio.c
521764        0x7F624         Unix path: /mnt/ss3_fw_source/esp-idf/components/esp32/./phy_init.c
540328        0x83EA8         Unix path: /home/xxt/work/code/esp32/ssc/components/smartconfig/./sc_sniffer.c
555736        0x87AD8         Unix path: /mnt/ss3_fw_source/esp-idf/components/freertos/./event_groups.c
556288        0x87D00         Unix path: /mnt/ss3_fw_source/components/libjansson/src/load.c
563384        0x898B8         PEM certificate
564808        0x89E48         SHA256 hash constants, little endian
578740        0x8D4B4         PEM RSA private key
578804        0x8D4F4         PEM EC private key
596892        0x91B9C         PEM RSA private key
598600        0x92248         PEM certificate
599812        0x92704         PEM RSA private key
601520        0x92DB0         PEM certificate
602736        0x93270         PEM RSA private key
```

While there wasn't any tooling available to meet our specific needs, the EspressIf ESP-IDF Programming Guide[5] and code in the esptool repository[6] ended up being all we needed to "roll our own."

These resources helped us to understand the overall layout of the flash, the boot process, and how the application code is stored — as well as its structure. Basically

the flash dump contains a bootloader, a partition table, and multiple partitions of varying types. A sample dump might look something like this:



ESP32 Firmware Layout

Of particular interest to us were the Factory and OTA partitions, as these are of type 'app' and, hence, contain application code. Multiple "application image" partitions can be present. These are created during the firmware build process, which is handled by the ESP-IDF. Prior to flashing firmware to a device, the application code is compiled into an ELF, and then esptool.py subsequently converts this into another binary format using an internal 'elf2image' function.

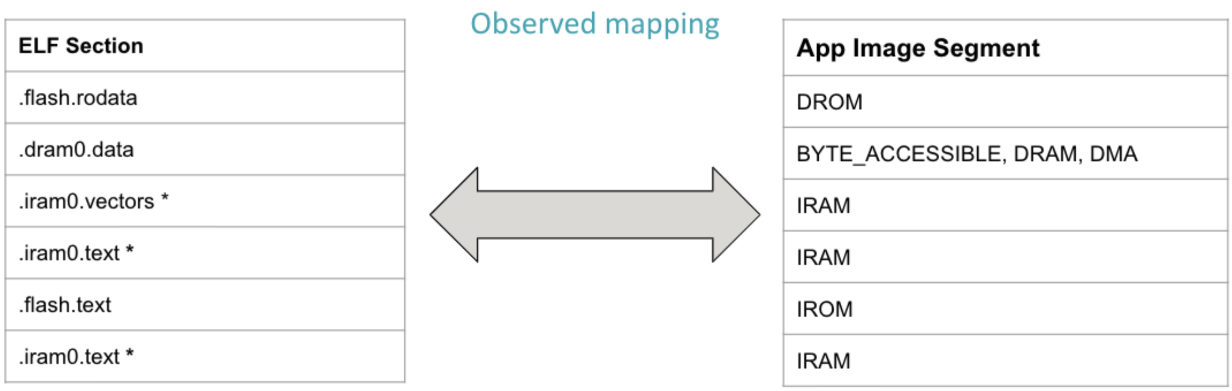
```
python esptool.py --chip esp32 elf2image --flash_mode dio --
flash_freq 40m --flash_size 4MB --elf-sha256-offset 0xb0 -o
/home/osboxes/esp/hello_world/build/hello-world.bin hello-world.elf
```

So the question became, "how do we extract the ELF from a firmware image?"

This drove us to analyze the 'elf2image' function in esptool.py, since we wanted to perform the reverse of that process. What we found is that specific ELF sections are selected to be included in a firmware application image, and others are left out. Only sections of type PROGBITS are included, so this excludes, for example, the

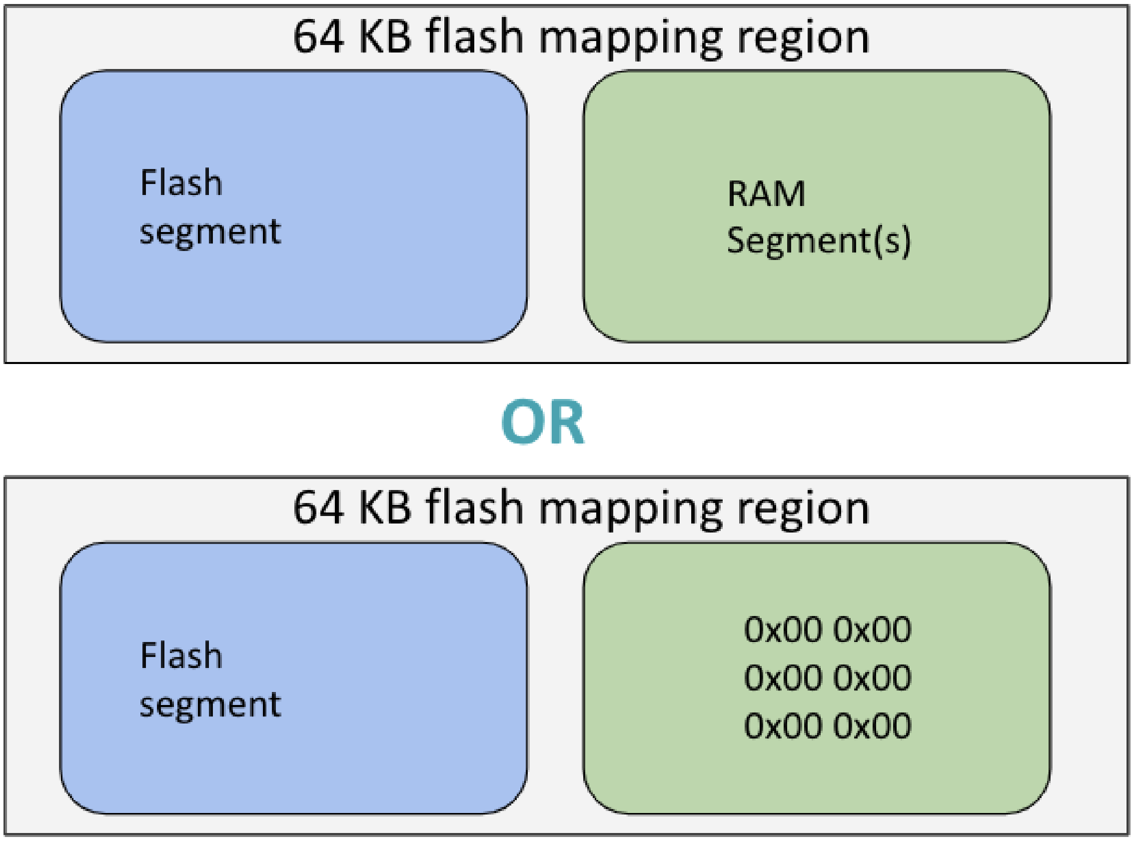
symbol table (.symtab). This means the extracted ELF won't contain any symbols, and this makes the analysis more challenging. Not to mention, the architecture is Xtensa.

Another piece of the puzzle was the ELF-section-to-app-image-segment mapping. We observed a consistent mapping in our testing (e.g. .flash.rodata maps to DROM).



ELF Section to App Image Segment Mappings

Additionally, sections are written to either RAM or flash segments depending on their address. For example, flash addresses range from 0x400D0000–0x40400000. Furthermore, flash segments might need padding due to a 64KB alignment requirement — padded with non-flash segment(s) or null bytes. There's more to the elf2image process, but we won't get into the nitty-gritty.



After analyzing the esptool.py elf2image code, we were able to construct our own function to extract an ELF. As I said, it performs the reverse of the build process. We have to parse the partition table, identify an "app" partition, and then create an ELF file using various segments from the image. Fortunately, we were able to reuse some of the code from esptool, but we also wrote some code from scratch — parsing logic, etc. The makeelf Python module was very helpful for constructing an ELF.

```
osboxes@osboxes:~/esp32_image_parser$ python3 esp32_image_parser.py create_elf flashdump/esp32_flashdump.bin -partition app0 -output flashdump/app0.elf
Dumping partition 'app0' to app0_out.bin
Writing ELF to flashdump/app0.elf...
```

Aside from being able to rip an ELF from a firmware image, our tooling can show partitions in an image, dump a specified partition to disk, and also dump NVS partition contents. If you're interested in diving a bit deeper and examining the finer details of our process, take a look at our talk from ShmooCon 2020 — Extracting an ELF from an ESP32[7]. The source code is available at https://github.com/tenable/esp32_image_parser[8].

```
Entry 5
Bitmap State : Written
  Written Entry 5
    NS Index : 2
        NS : nvs.net80211
    Type : BLOB
    Span : 4
    ChunkIndex : 255
    Key : sta.pswd
    Data (Blob) :
      Size : 65
      Data :
00000000: 6E 65 77 77 69 66 69 70  61 73 73 77 6F 72 64 21  newwifipassword!
00000010: 21 31 32 33 00 00 00 00  00 00 00 00 00 00 00 00  !123............
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000040: 00                                                .
```

Wifi Password stored in NVS

Unfortunately, the new tooling didn't help us to reveal any *critical* vulnerabilities in the SimpliSafe, but we did find a neat bug. Our analysis revealed some undocumented BLE functionality that allowed someone to pair up to the base station and modify the Wi-Fi network configuration. Basically, an attacker could tell the SimpliSafe base station to connect to his or her own network. Take a look at our research advisory[9] if you're interested.

Notes On Jamming

Many of you are aware of the sensor bypass attack via jamming using a cheap wireless remote off Amazon. This was publicized by LockPickingLawyer[18] on YouTube. They have improved the detection and alerting aspect somewhat, but it's still prone to false positives / false negatives. They are severely limited by the modulation scheme they are using (FSK[20] vs a more robust multi-carrier modulation scheme like OFDM[19] which is more resistant to interference, and jamming). They really can't fix any of this without releasing a new hardware revision (for sensors,

base station, and keypad) with better radios. Unfortunately, that will increase the cost of the system.

Conclusion

The SimpliSafe alarm system has very well-designed hardware, at a pretty low cost. The radio protocols are hard to attack outside of jamming. The security of the system is highly reliant on the security of PIC code protection, which is not 100% foolproof, especially for a well-funded attacker. With a bit more expensive hardware and bi-directional communication with the sensors/keypad to allow ephemeral keys to be exchanged, the system would a lot more difficult to attack, and resistant to attackers who gain access to the firmware, much like what we saw with the Ring alarm system[10]. Security through obscurity will never replace real system security.

References

[1] https://ioactive.com/remotely-disabling-wireless-burglar/
[2] https://www.bunniestudios.com/blog/?page_id=40
[3] https://www.youtube.com/watch?v=YIfc_jQAB9g
[4] https://www.youtube.com/watch?v=So4fzBzxbu8
[5] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/index.html
[6] https://github.com/espressif/esptool
[7] https://www.youtube.com/watch?v=w4_3vwN_2dI
[8] https://github.com/tenable/esp32_image_parser
[9] https://www.tenable.com/security/research/tra-2020-09
[10] https://medium.com/tenable-techblog/inside-amazons-ring-alarm-system-9731bc519974
[11] https://www.microchip.com/wwwproducts/en/PIC32MX170F512L
[12] https://en.wikipedia.org/wiki/Serial_Peripheral_Interface
[13] https://greatscottgadgets.com/hackrf/
[14] https://greatscottgadgets.com/yardstickone/
[15] https://github.com/atlas0fd00m/rfcat
[16] https://github.com/tenable/poc/blob/master/SimpliSafe/packet_decoder.py
[17] https://www.ti.com/lit/ug/swru295e/swru295e.pdf
[18] https://www.youtube.com/watch?v=UlNkQJzw4oA

[19] https://en.wikipedia.org/wiki/Orthogonal_frequency-division_multiplexing

[20] https://en.wikipedia.org/wiki/Frequency-shift_keying

Research Simplisafe Reverse Engineering Hardware Hacking

Tenable Research